Last modified 3/30/2011 by CFSworks

Introduction

This document describes Multiminer Protocol (MMP): a purely push-based TCP protocol for efficient Bitcoin mining. Since the purpose of this protocol is not to become the next web standard, this document doesn't go to great lengths to describe it very formally. I learn best by example, so I was sure to include plenty of those in this document. If there is any confusion, feel free to email me with questions at CFSworks@gmail.com

Basics

MMP is a simple line-based protocol with a message format based on that of IRC. That is, while IRC uses messages such as:

```
PRIVMSG #channel :Hello! <CRLF>
```

MMP uses messages like:

```
MSG :Hello! Thank you for connecting to my mining cluster! <CRLF>
```

Where <CRLF> denotes a CR, then LF character. (i.e. every message ends with "\r\n" as in IRC)

The arguments are space-separated. The last argument may start with a ':' character to indicate that it contains spaces. Therefore, no argument but the last may contain spaces. The commands are case-sensitive (so you should always send your commands in all-caps!) and any unrecognized commands are to be ignored. MMP's default TCP port is 8880.

The LOGIN command (client-to-server)

The LOGIN command is the first command the client should send. When it connects to the server, the server will (silently) wait for this command. The format of this command is:

```
LOGIN username :password
```

Note the use of the ':' -- the user might choose to use spaces in his password, so it's a good idea to include it. If the login succeeds, the server should immediately give the client work. Otherwise, the server should disconnect the worker. The server may send MSGs after the LOGIN command either to welcome the client to the mining server or explain why the client is being disconnected.

The MSG command (server-to-client)

This command may be sent by the server at any time (including before the client logs in) -- in its simplest form, this just instructs the client to print a message to its console. For example,

```
MSG :There are 3 workers connected, working at a total rate of 367.12 Mhash/sec
```

A message might start with "ERROR" in all caps. In all cases, this indicates a problem. The client may choose to render this message in a different color, or halt and ask the user if he wishes not to try to reconnect.

The BLOCK command (server-to-client)

This command exists to tell clients what the current Bitcoin block is. Though not required, it is highly recommended that this is sent to clients immediately after LOGIN and whenever the Bitcoin block number changes. If possible, send it before pushing new WORK to the client.

Example:

```
BLOCK 123456
```

The TARGET command (server-to-client)
Like the BLOCK command, only this one is absolutely required.
This must be sent before a WORK command, and only then. The target is not included in the WORK message because the target only changes every 2016 blocks, so it would be a waste of bandwidth to send it every time. Therefore, servers should only transmit a TARGET command when the target actually changes. The target follows the exact same syntax as a getwork() target. Example:
`TARGET ffffffffffffffffffffffffffffffffffffffffffffffffffffffff00000000`

The WORK command (server-to-client)
The server uses this to provide the client with an actual range of hashes to try. The client should use the most recent TARGET when checking the hashes. The server sends this message whenever the Bitcoin block changes (rendering previous work invalid anyway), immediately after LOGIN, and when the client asks for more.
The format of this command is `WORK data mask`, where data is an 80-byte hex encoded (i.e. 160-hex-character) string. The `data` is exactly as if you ran a Bitcoin getwork() and truncated the "data" field to 160 hex characters.
The mask is a number indicating how many bits of the nonce the client may change. 32 means the client has free reign of the entire nonce space (as in a getwork() request)
Notice: The mask indicates that the client may change the low N bits of the nonce, where nonce is interpreted as a **little-endian** integer (contrary to timestamp, version, etc.) unlike blockexplorer, etc.! The reason for this is so that the miner can increment the nonce in its native byte order. So, when the server issues:
`WORK 00000001...000000B0 28`
...it expects the client to try the range of hashes
`00000001...000000B0`-`00000001...FFFFFFBF` and NOT
`00000001...00000000`-`00000001...0FFFFFFF`
The server should clear the low N mask bits before sending the work to the client.

The RESULT command (client-to-server)
When the client finds a hash meeting the target, this is how it sends it in. It is identical in format to the 'data' field of a WORK command (see above) but with the last 4 bytes (8 hexadecimal characters) changed so that the hash meets the target. Example:
`RESULT 00000001...160...characters...long...38cf29a1`

The ACCEPTED command (server-to-client)
The server has accepted a RESULT and lets the client know by sending back an ACCEPTED command. It basically takes the RESULT command and replaces "RESULT" with "ACCEPTED" and sends it back down the line, like:
`ACCEPTED 00000001...160...characters...long...38cf29a1`

The REJECTED command (server-to-client)
The server rejected a RESULT. Works just like ACCEPTED.

The TIME command (server-to-client)
This indicates the maximum number of seconds the client may change work timestamp.
`TIME 120`
...this means that the client is allowed to move the timestamp forward, up to a maximum of 120 seconds. After which, it must renew its work with the MORE command.
If no TIME is sent, it is assumed to be 0

The MORE command (client-to-server)
The client is asking for more work. The server should give the client some more. This command takes no arguments, it's just "MORE" on a line on its own.
The client should not send any more MORE commands until the server gives it more work. That is, the client should only have one pending MORE command at a time. After the server receives one MORE command, it may ignore any subsequent MORE commands until it gives some work to the client.

The META command (client-to-server)
I saved this one for last, because it's probably the most fun. This is used by miners to share miscellaneous information with the server. The format is:
`META var :some value to store in var`

Some typical variables that the client may set are (note, these are case-sensitive!):
device: A brief description of the mining device. (`META device :ATI Radeon HD 5870`)
cores: The number of concurrent hashes the device runs (`META cores 320`)
rate: The current rate of the client in Khps. (`META rate 323177`)
(Server developers: Do not, I repeat, do **not** trust this. The client is only giving this as a statistic. You should already know how easy it is for clients to send false data.)
version: The version of the mining software. The version should *probably* follow this template:
`META version :Miner v1.2 by Author <OptionalEmail@example.com>`
--- OR ---
`META version :Miner r431 by Author`
The "by Author", version number, or both, may be omitted.
Also, it doesn't matter what you use for a version number, as long as it starts with 'r' or 'v'
name: A user-specified name used for the miner. (`META name :Home desktop 5870`)
os: The OS of the miner. Some examples:
`META os :Windows NT 6.1`
`META os :Linux v2.6.28-19-smp Ubuntu 9.04`
hostname: The hostname of the mining computer (`META hostname :CFSworks-PC`)
cpu: Brief information on the mining computer's CPU.

However, there are no restrictions on what data the miner may share with the server. I look forward to seeing things like
`META clockrate 853`
`META temperature 71.2`
`META system_uptime :4 days`

Just try to keep them under 1KB and don't try to store more than 16 or so on the server.

Note that values which are purely numeric are not preceded by a ':' character. This is purely a manner of style; including the ':' in a numeric variable is allowed, since it gets treated as a string by the server either way.

Appendix A: Writing a simple mining client

1. Connect to the server.
2. Send `"LOGIN username :password\r\nMETA version :MyMiner v1.0 by MyName\r\n"`
3. When the server sends MSG, print it on the console.
4. When the server sends TARGET, store it.
5. When the server sends WORK 0000... 20, do 2^20 work hashes.
6. When you start to get low on hashes, send "MORE\r\n"
7. Whenever the server sends TARGET, store the new value. Whenever it sends WORK, start on the new work immediately. Remember: It will send TARGET/WORK to inform you of more up-to-date work, so don't ignore the TCP connection while calculating hashes!
8. Reestablish connection to the server should the connection ever be dropped.

Appendix B: Pool commands

These commands are used by pools to share some pool-related information with the client. They are not required in the core MMP protocol, but it's a very good idea to implement them in pool servers.

The REWARD command (server-to-client)

A REWARD command looks like:

```
REWARD 4 0.03641728 123456 120
```

This command means, "your reward for round 4 is 0.03641728 BTC, when block 123456 gets 120 confirmations" (note: the 120 is not how many *more* confirmations it needs, but rather how many total; it should always be 120)

This means "you had 0.03641728 on round 4, but 123456 is now invalid":

```
REWARD 4 0.03641728 123456 0
```

The ROUND command (server-to-client)

Much like TARGET, it is sent before (and only before) a WORK command, to indicate the round that WORK counts toward.

The GETREWARD command (client-to-server)

Asks the server to please repeat a REWARD command, like so:

```
GETREWARD 4
```

Note, this is only used if the client wants to know about a round that happened while the client was offline (and therefore missed the initial REWARD messages) -- it is still the server's responsibility to send out REWARD commands whenever a round's status changes.

If the server receives a GETREWARD, and it doesn't yet know, it may disregard the GETREWARD (since it will send out a REWARD on its own once it does know)

Appendix C: Example connection
--> indicates "client-to-server" while <-- indicates the opposite

```
--> LOGIN user pass
--> META version :AwesomeMiner v20110213
<-- MSG :Welcome!
<-- MSG :This pool is currently running at 33.6 Mhps.
<-- MSG :If you need help, contact the admin at admin@example.com
<-- TIME 60
<-- BLOCK 123456
<-- TARGET 00000000...FFFF00000000
<-- ROUND 3
<-- WORK 000000014f28... 32
--> META rate 130527
--> META rate 130784
--> MORE
<-- WORK 000000014f28... 32
--> META rate 130278
--> RESULT 000000014f28...
<-- ACCEPTED 000000014f28...
<-- BLOCK 123457
<-- ROUND 4
<-- WORK 00000001edc0... 32
--> META rate 131018
--> MORE
<-- WORK 00000001edc0... 32
--> META rate 130978
<-- REWARD 3 0.02482736 123457 120
```

Appendix D: Enhancements to getwork()
Servers that provide a getwork() interface may include "mask" and "time" in the response:

```
{
    "midstate":"33b92ddbe57e2ae2777ea6ac32671a7259484eb8c2af23944717c6f87cd7b7d4",
    "data":"00000001b0f5753a46ef6048ddd1dda94fbc201e1eb11a59cdb22bec0000a37e0000000
0ec2272cafdf0e20a9891779d4efb9040f79cbec9d47138ccc226332dd215c7da4d9220291b00f33900
0000000000008000000000000000000000000000000000000000000000000000000000000000000000
0000000000080020000",
    "hash1":"0000000000000000000000000000000000000000000000000000000000000000000000000
80000000000000000000000000000000000000000000000010000",
    "target":"00000000000000000000000000000000000000000000039f3000000000000",
    "mask":32,
    "time":120
}
```